LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-636065

Active Measurement of the Impact of Network SwitchUtilization on Application Performance

M. Casas Guix, G. Bronevetsky

May 2, 2013

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Active Measurement of the Impact of Network Switch Utilization on Application Performance

Marc Casas, Greg Bronevetsky

Lawrence Livermore National Laboratory
7000 East Avenue
Livermore, CA, 94550

ABSTRACT

Inter-node networks are a key capability of High-Performance Computing (HPC) systems that differentiates them from less capable classes of machines. However, in spite of their very high performance, the increasing computational power of HPC compute nodes and the associated rise in application communication needs make network performance a common performance bottleneck. To achieve high performance in spite of network limitations application developers require tools to measure their applications' network utilization and inform them about how the network's communication capacity relates to the performance of their applications.

This paper presents a new performance measurement and analysis methodology based on empirical measurements of network behavior. Our approach uses two benchmarks that inject extra network communication. The first probes the fraction of the network that is utilized by a software component (an application or an individual task) to determine the existence and severity of network contention. The second aggressively injects network traffic while a software component runs to evaluate its performance on less capable networks or when it shares the network with other software components. We then combine the information from the two types of experiment to predict the performance slowdown experienced by multiple software components (e.g. multiple processes of a single MPI application) when they share a single network. Our methodology is applied to individual network switches and demonstrated on two real systems with different types of switches and compute node architectures.

1. INTRODUCTION

HPC applications demand very capable communication networks to support their high message and data volumes and/or tight synchronizations. Indeed, constraints on available network bandwidth or latency as well as network hotspots induced by specific communication patterns are often the key bottleneck that limit application performance [15, 2]. Looking into the future, it is expected that the computational capabilities of individual computing nodes will continue to rise faster than the capabilities of the networks that connect them [10]. This means that application performance will become increasingly bottlenecked on the capabilities of the network, making it even more imperative for application developers to optimize their applications taking network performance into account. Specifically, developers will need to (i) predict how their applications will perform on future systems with poorer network-to-node performance ratios and (ii) develop ways to assign computing work to available resources to effectively balance network communication and on-node computation. To achieve these tasks developers will require powerful tools to enable them to understand the interactions between their applications and the networks they run on and how these interactions ultimately affect application performance. Specifically, two directions of this interaction will need to be quantified for developers. First, tools must quantify how the application's communication utilizes the network and whether the application's needs are approaching the limits of the network's capabilities. Second, tools must measure how the capabilities of the network influence application performance and most importantly, whether the network is the application's performance bottleneck. These analyses must apply to both current and future systems, as well as to both static and highly configurable applications (e.g. where the space of possible configurations is too large to be explicitly enumerated and analyzed).

This paper presents a novel measurement-based approach to answer these questions. Unlike prior work based on simulation or tracing, our approach (i) experimentally measures an application's network use, (ii) quantifies it in terms of a simple metric based on resource utilization and (iii) identifies the relationship between available network capability and application performance. Given such measurements of multiple software components, such as full applications or their individual tasks, it is possible to predict how much their performance will degrade when they are executed on a less capable network (e.g. on a possible Exascale system), or concurrently on the same network where they contend for resources. This paper focuses on analyzing individual network switches and the compute nodes connected to them. We evaluate our approach on two different Infiband switches (QLogic and Voltaire) and compute node architectures (Xeon and Opteron).

The importance of network performance optimization has motivated significant research by the performance analysis

community. It can be divided into two categories: simulation and indirect measurement. The simulation approach, exemplified by tools such as SST [13], BigSim [20], Dimemas [12] or Venus [14] uses a detailed model of network hardware to account for the path of every message sent by each application node. Although these tools can accurately predict the performance of a particular application configuration on a particular network design, they have two limitations. First, the cost of using them can be high for many realistic large-scale applications since a full analysis requires (i) a large-scale application run where the details of its communication are stored in the parallel file system, followed by (ii) a detailed simulation of its communication. This is too slow to use for live application runs, although feasible for making projections to future systems.

Second, each simulation is valid for only one application configuration. To predict performance for a different assignment of application tasks to nodes or different distributions of work to tasks it is necessary to perform a simulation for this specific configuration. Since the space of possible permutations grows exponentially with the number of ways to configure the allocation of work to compute nodes, the simulation approach soon grows infeasible.

The indirect measurement approach is exemplified by tracing tools such as Vampir [9] and Paraver [11] as well as performance counter-based tools such as Tau [16]. In this approach various application regions are monitored to determine its communication structure, the amount of time it spends performing various operations and the number of events such as cache misses that occur during each operation. While these measurements directly capture ultimate application slowdowns as well as low-level hardware metrics, they can only enable indirect inference about how the properties of a network relate to application performance. For example, although such techniques can accurately measure the time from a message being sent to the time it is received, this time includes the cost of transferring the data from processor memory to the network card, the cost of network communication, the time the message spends waiting for the receiver node to post a receive buffer, etc. Using such inclusive measurements to infer key information such as network contention inherently loses accuracy. Further, this approach cannot be used to predict application performance on an alternate network or to predict how much contention for the network by multiple software components can affect their performance.

2. APPROACH

This paper presents a new approach to measure the relationship between network capability and application performance. Our basic insight is that this relationship should be modeled as the application consuming a resource provided by the network. As more of this resource is available, the application runs monotonically faster, with reduced improvements as application performance becomes bottlenecked on other resources. Further, if multiple software components (entire applications or individual tasks such as processes, threads or Charm++ chares [7])) run concurrently on the same network, they will share its resources. This sharing can be modeled as one component consuming some amount of network resources, making it unavailable to others and thus causing them to behave as if they were running on a less capable network. The heart of this idea is a "performance relativity" principle, that "from the perspective of software components less capable networks behave very similarly to networks that are partially utilized by other software components". This principle enables two novel measurement techniques that can answer the above questions:

Impact experiments measure a software component's use of the network based on the latency of additional messages sent over the network while the component runs. These measurements directly quantify the network's ability to carry application communication and can be used to determine whether the network is congested and measure how close the application is to fully utilizing the network. The additional messages are triggered by extra tasks running on dedicated cores and they do not impact applications' performance as the extra load is very low.

Compression experiments measure the relationship between network capability and a software component's performance. The component is executed concurrently with a micro-benchmark that runs on cores connected to the same network and sends varying volumes of communication. As the effective network capability is varied we observe the component's resulting performance, which corresponds to how it will perform on less capable networks or when more software component are executed on the same network.

Finally we present a technique that combines the two measurements to predict the performance degradation that a given combination of software components would suffer when executed concurrently on the same network. It is based on a common metric that quantifies available network capability by modeling the network as a mathematical queue [18] and using data from Impact measurements to compute the fraction of the queue that is utilized. By measuring the network capability that is left available while a given application or the Compression benchmark executes we can estimate the effect of multiple concurrent software components on each other as they share a network. The experimental and analytic procedures presented in this paper are focused on single-switch networks that connect multiple computing nodes.

Our approach improves upon the state of the art in network performance modeling and measurement in the following ways:

i) Impact experiments of network utilization and contention are significantly faster than similar analyses performed inside simulators and apply to real physical networks for which precise models may not exist due to intellectual property restrictions. Further, unlike indirect measurement techniques, Impact experiments directly probe the network's ability to carry out the application's communication requests. Since they focus on just the network and quantify its effective capabilities in terms of a generic queue-oriented metric, these experiments provide a simple and unfiltered view onto this resource.

ii) Compression experiments and Performance Degradation analysis make it possible to relate application performance

to network capability. While simulators can predict the performance of specific workloads on specific networks, a separate simulation run is required for each configuration. As the number of configuration options increases (e.g. number of atoms per core or the assignment of software components to different cores), the number of such experiments rises exponentially. In contrast, our approach scales linearly with the number of software components that must be measured independently.

iii) Our techniques are enabled by a new queue-oriented metric for measuring network utilization. This metric has the key property of varying monotonically with the utilization of the physical network as well as with the resulting application performance.

This paper is structured as follows: Section 3 presents the experimental setup used and briefly describes the set of applications we use in our experiments. Section 4 describes Impact measurements and how they can be used to feed an analytical model based on queue theory. Section 5 describes Compression measurements, details how they Interact with impact measurements and shows the performance analyses they enable for real applications. Section 6 presents and validates our methodology for predicting performance of real complex workloads that share the same network switch.

3. EXPERIMENTAL DESIGN

The experiments in this paper were conducted on the Cab and Hera clusters at the Lawrence Livermore National Laboratory. Cab is composed of 1,296 compute nodes, each of which includes two 8-core 2.6Ghz Intel Xeon E5-2670 processors with 32GB of RAM. Hera has 864 compute nodes, each one has four 4-core 2.3Ghz AMD Opteron 8356 processors with 32GB of RAM. The networks on Cab and Hera are QLogic Quad-data-rate and Voltaire Double-data-rate Infiniband, respectively, organized into a two-level fat tree. This paper focuses on the bottom-level switches of both networks. In Cab they are QLogic 12300, with 36 ports, of which 18 are used to connect compute nodes and 18 connect to the second-level switch. These switches provide approximately 1μs of network latency and 5GB/s bandwidth. On Hera they are Voltaire 9024, with 24-port switch, of which 12 connect to compute nodes and 12 to the second-level switch. They provide approximately 3-5 μ s latency and 2.4 GB/s bandwidth. Each experiment on Cab and Hera was run on groups of 18 or 12 nodes, respectively, connected to a single bottom-level switch and our results are thus not affected by interference from other applications running on the same cluster.

Our experiments focus on the following applications:

MILC [3] - The MIMD Lattice Computation, a Quantum Chromodynamics simulation with lattice size nx=16, ny=32, nz=32, nt=36.

FFTW [5] - Fast Fourier Transform library that uses hierarchical composition of multiple FFT algorithms, applied to perform a 2D transform of a 2000x2000 matrix.

Lulesh [1] - The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics simulation that is a materials science proxy application, executed on a 22x22x22 cube domain.

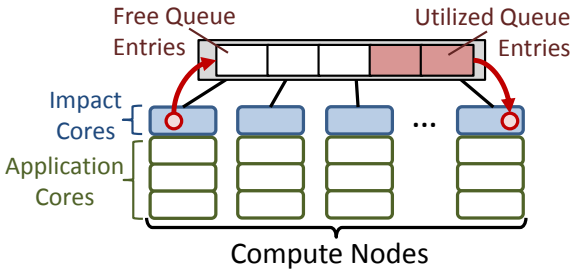

Figure 1: Impact Interference

MCB [4] - A continuous energy Monte Carlo Burnup Simulation Code for studying nuclear waste transmutation systems, executed on 3,000,000 particles.

4. IMPACT ANALYSIS

4.1 The Impact benchmark

The basic idea behind Impact experiments is that the degree to which an application utilizes a network switch can be measured in terms of how well the network can service additional communication requests. Application messages are broken up into multiple small (few KB) packets and sent to the network switch. As illustrated in Figure 1, packets from one compute node arrive on one port of the switch, propagate through its internal circuitry and exit via the port of its destination node. Since the execution time of communication operations depends on the transit time of each packet, the distribution of these times captures the network's effective capability that is available to applications. Further, when some software component is already utilizing the network, the difference between this distribution during the component's execution and the same on an unloaded network measures the amount of network capability the component uses up and leaves unavailable to others.

We measure the latency of packets through the network switch using the simple micro-benchmark listed in Figure 2, which we denote ImpactB. Compute nodes on the same switch are paired and execute a ping-pong data exchange where the process with the even rank sends a message, the process with the odd rank receives it and replies with another, which is finally received by the initial process. The entire exchange is timed by the initiator process to determine the average latency of the two messages, which are set to be 1KB in size to ensure that they are communicated via a single network packet. Each ping-pong exchange is separated by a 100ms sleep to minimize ImpactB's effect on the executing application.

4.2 Measurements

Figures 3 and 4 shows the distribution of message latencies observed on Cab and Hera, respectively, both when executing ImpactB on an unloaded switch and when ImpactB is executed concurrently with our target applications. In these experiments the processes of ImpactB and the target application were spread over all the compute nodes connected to

```
while(1) {
  if( my_node%2 == 0 && my_node!=n_nodes-1 ) {
        PMPI_Isend(... , (my_rank+tasks_per_node)%(n_nodes), ... , &request);
        PMPI_Irecv(... , (my_rank+tasks_per_node)%(n_nodes), ... , &request2);
  } else if ( my_node%2 == 1 ) {
        PMPI_Irecv(... , my_rank-tasks_per_node , ... , &request);
    PMPI_Isend(... , my_rank-tasks_per_node , ... , &request2);
   }
   PMPI_Wait(&request , status);
   PMPI_Wait(&request2 , status);
   usleep(100000);
}
```

Figure 2: Pseudo-code of the ImpactB micro-benchmark

the switch. On both Cab and Hera 2 ImpactB processes was executed on every node. Since Cab's nodes have 2 sockets, an ImpactB process was run on each socket. Since Hera's nodes have 4 socket, 2 of them were assigned a single ImpactB process and the other 2 were assigned none.

The application processes were executed on the remaining cores. On Cab we executed 4 processes of MILC, FFTW and MCB on each socket, 8 per node for a total of 144 across all the 18 nodes connected to a switch. On Hera, we map 2 processes of MILC, FFTW and MCB per socket, 8 per node for a total of 96 processes across all the 12 nodes on a switch. On Cab Lulesh, which needs a cubic number of processes, was run on 16 nodes, utilizing 2 cores on each socket, for a total of 64 MPI processes. On Hera we ran Lulesh on 8 nodes, mapping 2 processes on each node, 8 per node for a total of 64 MPI tasks.

The remaining cores were left idle in these experiments. This assignment of application processes to cores was used to simplify the presentation of the performance prediction experiments in Section 6, which discusses performance prediction for multiple concurrently-executing applications.

The Cab data shows that when the switch is not loaded, packet latency is 2.5μs on average, with many packets taking a little less or more time and a few packets taking significantly longer. When the applications are running the latency distribution shifts. The execution of FFTW and MCB on Cab shifts 20% of packets from taking approximately 2.5μs to take more than 5μs. In contrast, the primary effect of Lulesh and MILC is to shift the mode of the distribution to the right, close to 5μs. Further, while Lulesh didn't cause an increase in the fraction of packets with very high latency, with MCB this effect was strong.

The results on Hera are notably different. First, the latency distribution on an unloaded switch is significantly broader. Whereas most packets on Cab have latency <5μ s, on Hera fewer than 50% of packets fall into this category. The remaining packets cover a range of latencies that is approximately twice as wide. Interestingly, although FFTW has a significant impact on packet latency on Cab, it has almost no impact on Hera's switches. The other three benchmarks all shift the mode of the distribution from 2.5μs towards higher latencies, with MCB moving it to 6μs, MILC to 10μs and Lulesh to 17μs.

The differences between the two networks are striking and indeed, since we do not have a simulator for these proprietary switches, we do not have a clear explanation for these differences. Importantly, this lack of deeper insight into network internals, which is the focus of some performance of some performance analysis techniques, is irrelevant for our methodology. As we show in Section 6, by using these direct measurements of application behavior on the different switches we can make quantitative prediction of application slowdown in different utilization scenarios without knowing anything about the internal details of the switches or the applications that use them.

4.3 Queue Theoretic Switch Metric

While packet latency distributions can provide some insight into the effective capability of the switch, they are too complex to measure switch capability as a resource. This is because they do not vary monotonically with application performance since it is not clear whether one distribution represents more or less network utilization than another (e.g. compare Lulesh and MCB's distributions). However, they can be used to extract the appropriate metric by modeling the behavior of a switch as a mathematical queue and leveraging the results of queueing theory (QT) [18] to infer the state of this queue based on its observable behavior (the packet latencies).

QT is a math discipline that studies queues in terms waiting times and line lengths and has been successfully applied to model call centers, factories or city traffic. We represent the real switch as a queue by considering that each packet arrives at one switch port, is processed by internal switch circuitry and then departs via another port. As Figure 1 illustrates, when the packet arrives at this queue other packets may already be waiting in the queue to be routed, forcing the packet to wait until these packets are processed. The length of the queue inside the switch depends on the pattern of packet arrival times at the switch. We specifically, we use the M/G/1 queue model to represent switch routing logic. This model assumes that:

- The size of the queue is unbounded,
- Queue arrival traffic is modeled by a Poisson Process [8]: (i) the time between two consecutive events follows an exponential distribution with parameter λ and (ii) each inter-arrival time is independent of the prior ones,

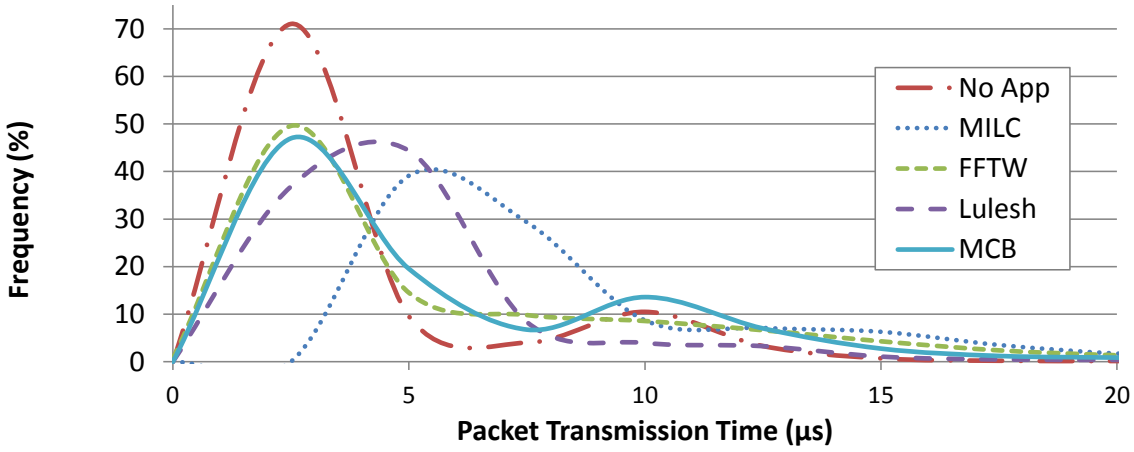

Figure 3: Packet latency histogram on Cab

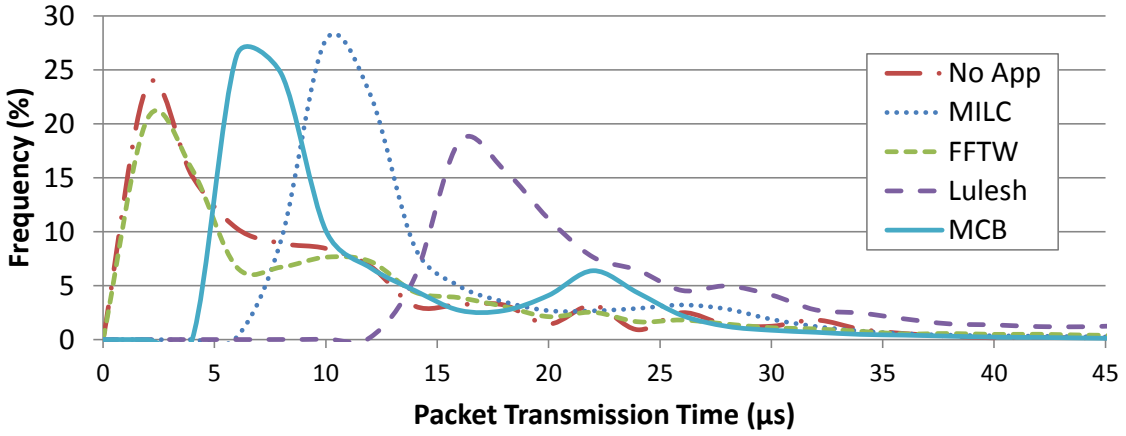

Figure 4: Packet latency histogram on Hera

- Packet service times are modeled by some probability distribution, and
- Packets are processed by a single server.

QT defines the utilization of a queue as the proportion of its entries that are used by the arriving traffic. Utilization ρ can be expressed as the rate $\frac{\lambda}{\mu}$, where λ is the mean rate of packet arrivals and μ is the mean rate of packet service times. If $\rho \geq 1$ then the queue's waiting time will grow, which implies that the switch will be contended and application performance will degrade significantly. Parameters λ and μ must be known to measure ρ. μ is a hardware parameter that is measured by sending multiple individual packets into an idle switch and measuring their minimum latency. λ is an application specific parameter that can only be directly measured by using switch counters, which are not available in general as they require root privileges. However, λ can be computed via the the Pollaczek-Khinchine formula [6]:

$$W = \frac{\rho + \lambda\mu Var(S)}{2(\mu - \lambda)} + \mu^{-1} \quad (1)$$

Where W is the total average time spent by packets in the queue either waiting and being serviced and $Var(S)$ is the variance of the service times. Since utilization $\rho = \frac{\lambda}{\mu}$, we can write the formula as:

$$W = \frac{\frac{\lambda}{\mu} + \lambda\mu Var(S)}{2(\mu - \lambda)} + \mu^{-1} \quad (2)$$

which can be transformed to compute λ as follows:

$$\lambda = \frac{2 - 2W\mu}{-2W + \frac{2}{\mu} - \mu Var(S) - \mu^{-1}} \quad (3)$$

$Var(S)$ can be computed from the single-packet experiments on an idle switch and importantly, W is just the average latency of the packets communicated by ImpactB while the target application runs. Since utilization $\rho = \frac{\lambda}{\mu}$, we can compute it by using the the above formula given the parameters obtained through ImpactB measurements.

Table 1 presents the measured queue utilization of our target applications. The data shows that none of the applications

	FFTW	Lulesh	MCB	MILC
Cab	51%	37%	48%	70%
Hera	82%	38%	40%	61%

Table 1: Application Queue Utilization

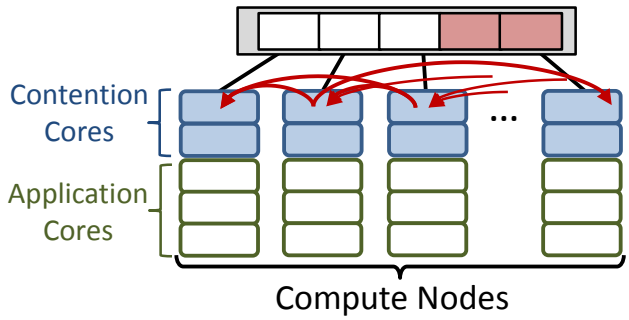

Figure 6: Interference by the CompressionB micro-benchmark

fully saturate the network, although MILC comes close at 70%. The queue utilization of FFTW and MCB is similar at 51% and 48%, respectively, just like the packet latency distributions they induce. However, the queue utilization of Lulesh is actually lower, at 37% since even though it shifts the mode of the distribution by a few μs, fewer packets experience significant delays.

5. COMPRESSION ANALYSIS

5.1 The Compression benchmark

Compression experiments measure the relationship between the network capability available to a software component and its performance by incrementally reducing network capability and observing the effect of this on performance. Since it is not possible to adjust the properties of real switches and network simulations are expensive, we use the performance relativity principle (reduced network capability affects application performance similarly to resource sharing) to simulate reduced network capability via software interference. We execute the target software component on a subset of the available cores. On the remaining cores we execute the CompressionB micro-benchmark, the pseudo-code for which is listed in Figure 5. CompressionB is executed on the same number of cores on each node, where processes running on the same core ID on different nodes are organized in a 1-dimensional communication ring. As illustrated in Figure 6, in each iteration every CompressionB process sends a 40KB message to P partner processes that precede it in the ring (all processes in its ring are on different nodes) and receives the messages sent by the P succeeding processes. After M messages have been sent in this way, the benchmark sleeps for B cycles, waits for all the `MPI_Irecvs` and `MPI_Isends` to complete, and repeats the communication pattern.

Various settings of parameters P, M and B degrade network capability to different extents. Thus, by performing multiple experiments where a different configuration of CompressionB is executed concurrently with a target software component it is possible to measure the degradation in the component's performance on less capable switches. This corresponds to future systems where the network performance is poorer relative to processor performance, as well as scenarios where more application work is assigned to and contends for the same network.

5.2 Switch Utilization of CompressionB

To quantify the fraction of switch capability that various configurations of CompressionB use, we run it together with ImpactB just like any other software component ImpactB may measure. This measurement makes it possible to relate performance degradation to the fraction of switch queue capability removed by CompressionB. The result is a high-level description of application performance in terms of a generic measure of network capability, the queue utilization fraction.

Our CompressionB+ImpactB experiments are executed using the same configuration as above, where we map 1 ImpactB and 1 CompressionB process on each socket, for 2 ImpactB and 2 CompressionB tasks per node. Figure 7 shows the range of different queue utilization percentages that can be achieved by all the considered variants of CompressionB when run on Cab. Parameter P, the number of partner processes, takes values 1, 4, 7, 14 and 17. Parameter B, the number of cycles the benchmark sleeps, has values $2.5E4$, $2.5E5$, $2.5E6$, $2.5E7$. Finally, parameter M, the number of messages sent in each round of communication, is either 1 or 10.

The data shows that main determinant of switch queue utilization is the number of cycles the benchmarks sleeps, with utilization decreasing with longer sleeps. Further, utilization rises with increasing partner counts and message counts. The effect of partner count is strongest for longer sleep times while the effect of message count is strongest for shorter sleep times. In total, we consider 40 different input configurations, which allow us to cover switch queue utilizations between 26% and 92%. The broad range of queue utilizations provided by these configurations enables us to precisely evaluate applications performance degradations due to reduced switch capability.

We have also run similar experiments on Hera, considering 1, 4, 6, 8 and 11 partner processes and setting the other parameters to the the same values as we mention above. The main determinant of switch utilization is again the number of cycles the benchmark sleeps. Also, we can see how the switch is already using 75% of each capacity when 10 messages are sent and parameter B is $2.5E6$, while the Cab switch uses between 30 and 40% of each capacity to handle the same traffic. That clearly indicates that Hera switches have less capacity in absolute terms than the ones installed in Cab.

5.3 Application performance impact due to reduced network capability

We used CompressionB to measure the relationship between available network switch capability and the performance of our target applications. Each experiment used the same configuration as in Section 4, with 2 CompressionB processes per node. On Cab we assigned 1 CompressionB process per

```
while(1) {
  for(partner=0; partner<P; partner++) {
    for(mesg=0; mesg<M; mesg++) {
      // Receive from same core ID on succeeding node
      PMPI_Irecv( ... , (my_rank+tasks_per_node*(partner+1))%comm_size, ... );
      // Send to same core ID on the preceding node
      PMPI_Isend( ... , (my_rank-tasks_per_node*(partner+1)+comm_size)%comm_size, ... );
    }
    usleep(B);
  }
  MPI_Waitall( ... );
}
```

Figure 5: Pseudo-code of the CompressionB interference micro-benchmark

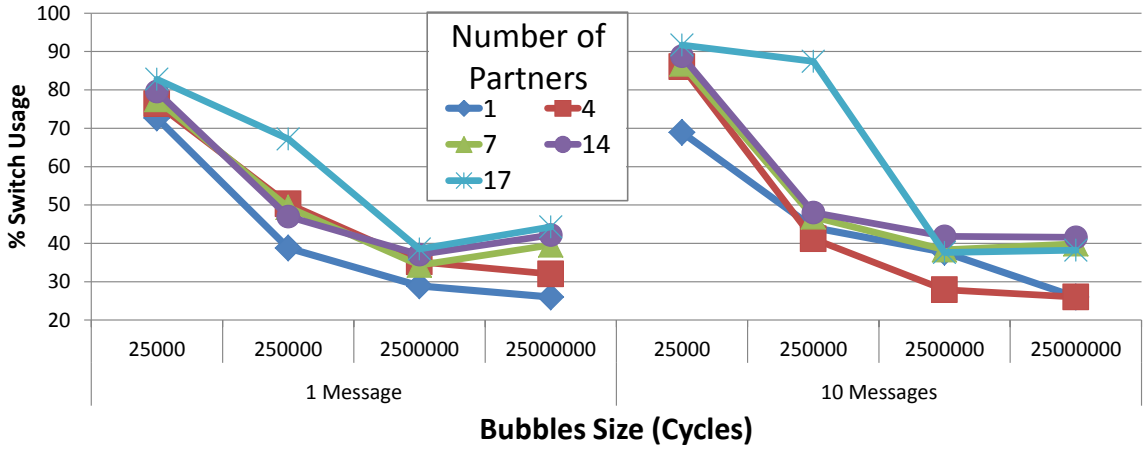

Figure 7: Switch usage compression benchmark on Cab

socket, while on Hera we assigned 1 CompressionB process to 2 of the sockets and did not assign a CompressionB process to the other 2 sockets. The other cores were assigned to the application or left idle. Figures 9 and 10 show the percentage performance degradation on Cab and Hera, respectively, of FFTW, Lulesh, MCB and MILC (y-axis) as the percentage of switch utilized by CompressionB changes across its full range (x-axis) due to the use of different configuration parameters. Performance degradation is computed as $\frac{\text{Run time with interference - Run time with no interference}}{\text{Run time with no interference}}$. The left sub-figure shows the data with a linear y-axis and the right sub-figure uses a logarithmic y-axis. For each application we fit the data points with the best linear approximation to highlight the overall trend of the results.

Reducing switch capability has the most effect on FFTW, which runs more than 50% slower on Cab when even 40% of the switch queue is utilized and up to 250% slower as utilization reaches 92%. MILC is also significantly affected, running approximately 20% more slowly on Cab at 40% queue utilization and over 100% more slowly at 92% utilization. This is because both applications are very sensitive to the latency of messages, meaning that if on average the queue is 40% full the stochastic nature of packet arrivals means that there are many packets that arrive when the queue is very long. Recall that the packet latency distributions shown in Figure 3 have some high latency packets even when the switch is idle. When the switch is partially utilized the fraction of high latency packets can become considerable, significantly degrading the performance of FFTW and MILC.

In contrast, Lulesh and MCB are significantly less affected on Cab by reduction in switch capability. The performance of Lulesh degrades by 8% at 50% queue utilization and 15% at 92% utilization. MCB is almost completely insensitive to queue utilization, slowing by no more than 3.5% across the full utilization range.

Figure 10 shows the results from the same experiments on Hera. The data shows that the performance of each application degrades similarly on this platform. One major difference is that FFTW's performance doesn't degrade as significantly on Hera as on Cab for switch utilizations below 90%. In contrast, MILC's performance is even more significantly degraded for these intermediate utilization levels than it is on Cab. MCB and Lulesh as as insensitive to interference as on Cab, except at almost 100% interference where Lulesh performance degrades by 20%.

The above experiments make it possible to estimate the performance of software components when executing on switches with different capabilities. Specifically, to focus on a particular scenario it is necessary to choose the queue utilization fraction that corresponds to the removal of the given amount of switch capability and run the application with CompressionB configured to emulate this utilization fraction.

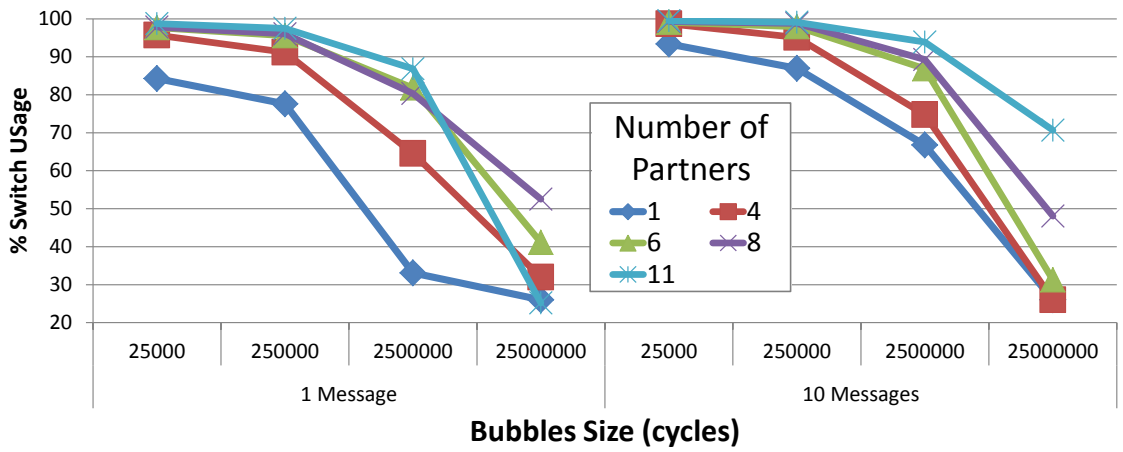

Figure 8: Switch usage compression benchmark on Hera

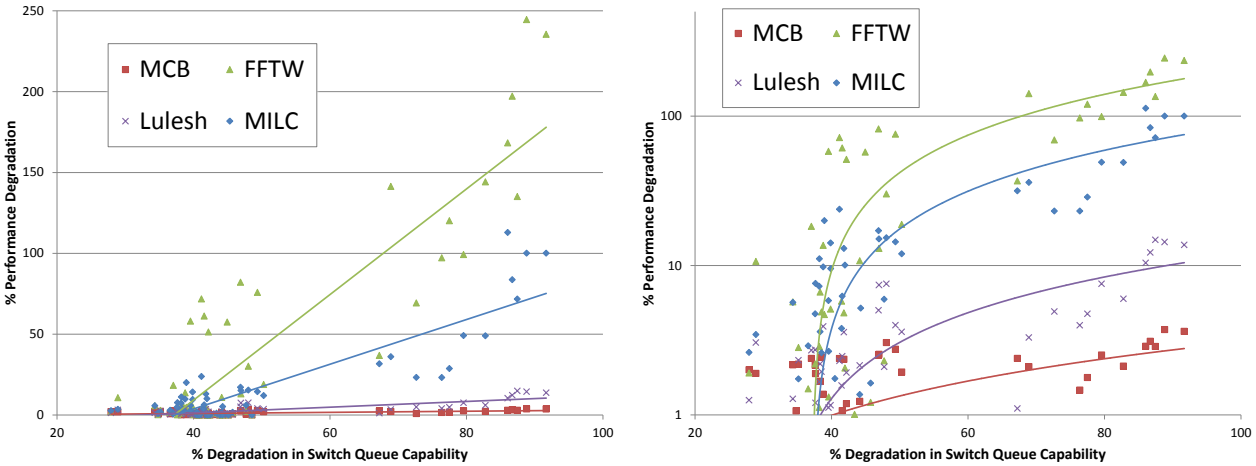

Figure 9: Performance degradations suffered by real application in terms of switch queue utilization by CompressionB availability in cab. In the left, the y-axis is expressed as a linear scale. In the right, it is expressed as a logarithmic scale.

6. PREDICTION

Section 4 showed how to measure the application's utilization of switch resources by timing the latencies of individual packets (the ImpactB benchmark) and inferring from this the fraction of the switch's internal queue that is utilized. Section 5 then presented a way to emulate switches with reduced capabilities by running concurrently with the application an interference workload (the CompressionB benchmark) that reduces the amount of switch capability available to the application. Further, this section described a way to quantify the amount of interference induced by CompressionB in terms of the same queue utilization metric. We now show how to combine these two experimental techniques to make quantitative predictions about how the performance of multiple software components (application tasks or entire applications) will suffer when they are executed concurrently on the same switch. Critically, our approach makes it possible to make predictions for new combinations of software components (number of combinations grows exponentially with the number of components and polynomially in the number of their configurations) based on experiments performed on each component in isolation from the others (grows linearly with the number of components).

Consider software components A and B. Impact experiments conducted on A and B will result in quantities $U_A\%$ and $U_B\%$ that measure the fraction of the switch queue each component utilizes. Compression experiments on these components produce mappings p_A and p_B that map queue utilization fractions to the performance degradation in each component. We then use the configurations of CompressionB that also utilize $U_A\%$ and $U_B\%$ of the switch queue to model the effects of A and B, respectively on other software components with which they share a switch. We thus predict the performance degradation of A when executed concurrently with B to be $p_A(U_B)$. Specifically, this means that A's performance will degrade as much when sharing the switch with B as it did when it shared the switch with the configuration

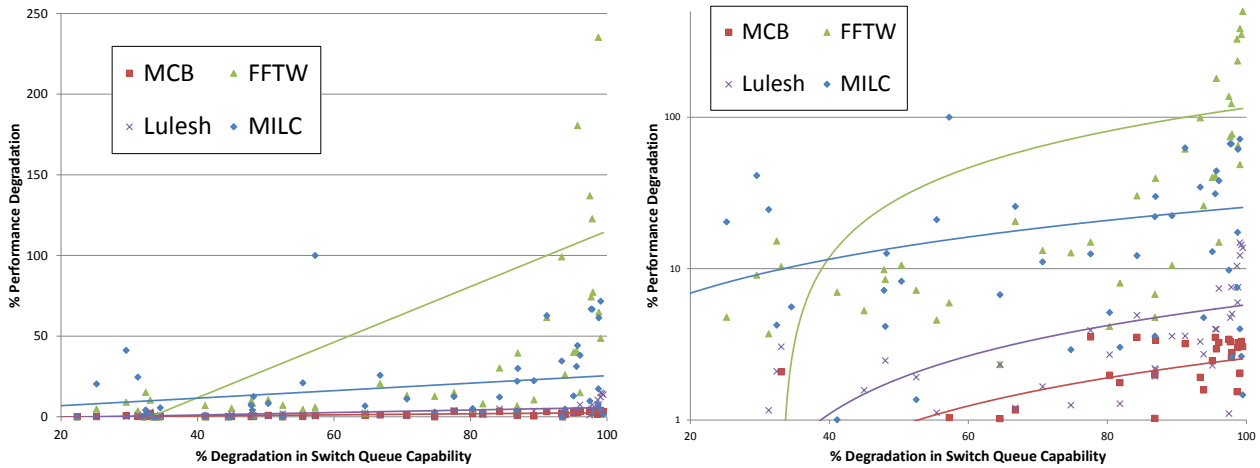

Figure 10: Performance degradations suffered by real application in terms of switch queue utilization by CompressionB availability in cab. In the left, the y-axis is expressed as a linear scale. In the right, it is expressed as a logarithmic scale.

of CompressionB that utilizes the same fraction of the switch queue as B does. The converse prediction is made for B. This analysis can be performed for any combination of application tasks, their configurations (e.g. number of molecules simulated or the size of their communication stencil) or even multiple concurrently executing applications.

We evaluated the accuracy of this prediction algorithm by running pairs of our target applications concurrently on the same switch to observe whether the model correctly predicts how much they degrade each other's performance. In these experiments each application was executed using the configurations used in the experiments reported in Sections 4 and 5. Specifically, for the experiments run on Cab with MILC, FFTW and MCB we ran 4 processes on each socket, for a total of 144 processes on the 18 dual-socket nodes connected to one switch. Since Lulesh must run on cubic numbers of processes, we ran 2 Lulesh processes on each socket on 16 nodes, for a total of 64 processes. In our Hera experiments, we ran 2 processes of MILC, FFTW and MCB on each socket for a total of 96 processes on the 12 4-way nodes connected to one switch. We ran Lulesh ran 2 processes on each socket across 8 nodes, for a total of 64 processes. This process mapping utilizes at most half the available cores, leaving enough cores for two applications to run concurrently without sharing cores. Our experiments include combinations where two copies of a single application run concurrently on the same nodes and switch, as well as combinations where two different applications execute together. The former evaluates our model's accuracy on the use-case of HPC capability computing where different amounts of a single application's work may be assigned to a single switch. The latter accounts for the use-cases more typical in cloud computing or HPC capacity computing where multiple applications may share a single switch, as well as applications such as ddcmd [17] that run processes dedicated to molecular dynamics and processes for FFT computations concurrently on different nodes on the same network.

Figures 11 and 12 present the results of the 10 experiments (4 experiments where each application was run with itself and 6 experiments for different application pairs) executed on Cab and Hera, respectively. The y-axis shows the percent performance degradation of each application in each pairing, while the x-axis shows each pairing $X - Y$. Since experiments where two different applications are executed concurrently result in two different performance degradations they are listed separately on the x-axis, for a total of 16 different degradation measurements. Specifically, each x-axis label $X - Y$ should be read "performance degradation on application X when it runs concurrently with application Y".

Our results show that overall our model has very good predictive capability on both systems. It clearly separates the pairings that induce little performance degradation from those that induce significant degradation. For the low-degradation combinations the primary errors were that (i) the model predicts zero degradation while in reality performance degrades by 3%-5%, (ii) it predicts a degradation that a few percent higher or lower than reality (Cab: MILC with Lulesh and MCB, Lulesh with MCB and MCB with MILC; Hera: FFTW with MILC, Lulesh with FFTW and MCB with FFTW) or (iii) the model predicts a notable degradation where in reality it was small, as for MILC when co-executing with MCB on Hera.

For combinations where performance degraded significantly the model's predictions were generally close to real observations. The only significant error was that the model predicted that the performance of FFTW when co-executing with MILC on Cab would degrade significantly more than it actually did. Although the model mis-predicted the exact amount of degradation, it did correctly identify that in this case performance degradation would be notable, since it was actually $> 10\%$.

7. CONCLUSION

In this paper we have shown the usefulness of proactive measurements to analyze applications' consumption of switch re-

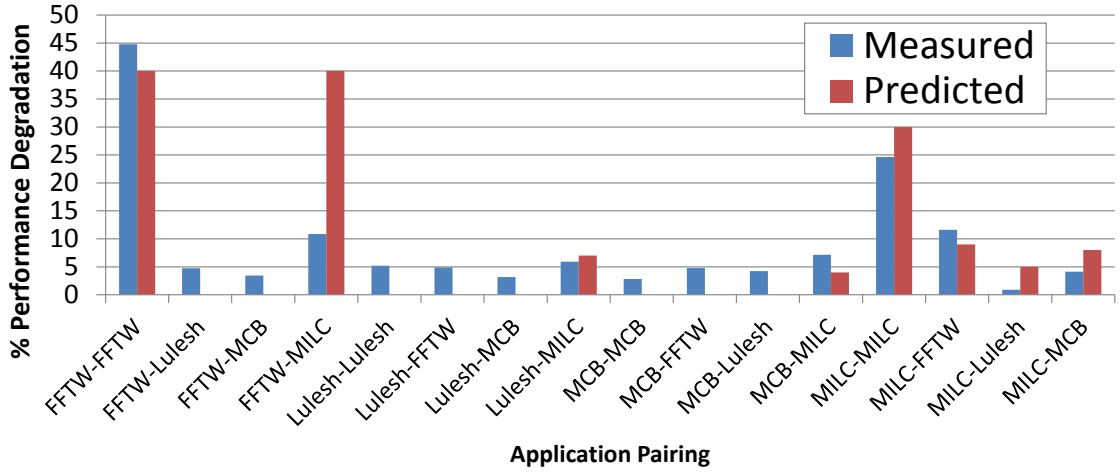

Figure 11: Performance predictions for combined workloads in Cab

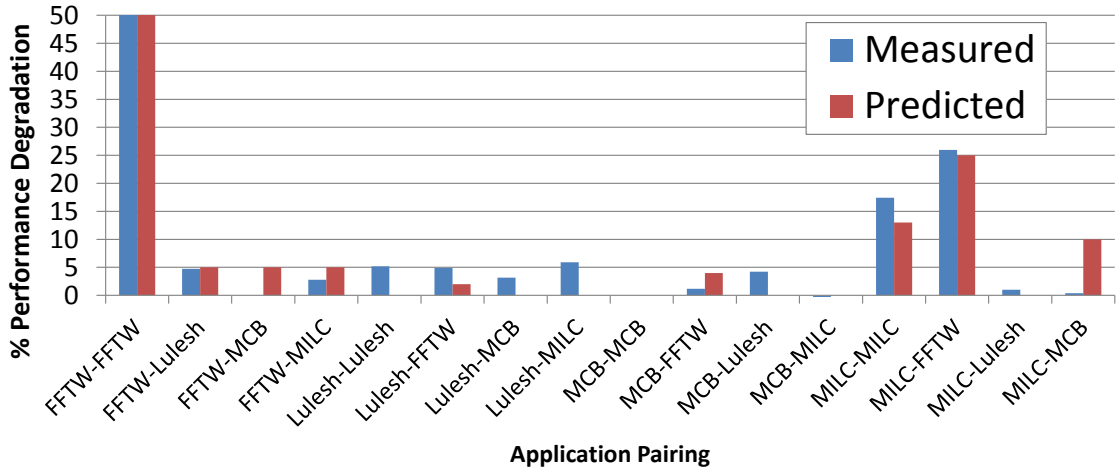

Figure 12: Performance predictions for heterogenous workloads in Hera

sources and to predict performance degradations when those resources are shared with other workloads. This a very important problem since high performance computing infrastructures typically run several applications on the same time, all of them sharing the network. Our technique uses two interferences that inject extra network workload. The first detemines the fraction of the network that is utilized by a software component (an application or an individual task) to figure out the existence and severity of network contention. The second aggressively injects network packets while a software component runs to evaluate its performance on networks with less capacity or when it shares network resources with other software components. We then combine the information from the two types of experiment to predict the performance slowdown experienced by multiple software components (e.g. multiple processes of a single MPI application) when they share a single network. We have also validated our approach by comparing the predictions we get through our modeling and measurement techniques with real measurements obtained when two applications run together on the same switch. The overall model is fully generic and can be applied to memory hierarchies as well file systems by considering combinations of multiple queues and caches via extensions of network tomography techniques [19]. A new generation of performance analysis tools and techniques focused on applications' resource consumption rather than simple low-level measurements that are not always actionable can be developed from the ideas explained in these paper.

8. REFERENCES

[1] Hydrodynamics Challenge Problem. Technical Report LLNL-TR-490254, Lawrence Livermore National Laboratory.

[2] *Scientific Application Performance on Candidate PetaScale Platforms*. IEEE Computer Society, 2007.

[3] G. Bauer, S. Gottlieb, and T. Hoefler. Performance Modeling and Comparative Analysis of the MILC

Lattice QCD Application su3_rmd. In *CCGRID*, pages 652–659, 2012.

[4] J. Cetnar, J. Wallenius, and W. Gudowski. MCB: A Continuous Energy Monte-Carlo Burnup Simulation Code. In *Actinide and Fission Product Partitioning and Transmutation*, 1999.

[5] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[6] J. Haigh. *Probability Models*. Springer, 2002.

[7] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[8] J. Kingman. *Poisson Processes*. Oxford University Press, 1993.

[9] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The Vampir Performance Analysis Tool-Set. In M. M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, editors, *Parallel Tools Workshop*, pages 139–155. Springer, 2008.

[10] P. Kogge. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical report, DARPA IPTO, September 2008.

[11] J. Labarta. New Analysis Techniques in the CEPBA-Tools Environment. In M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, editors, *Parallel Tools Workshop*, pages 125–143. Springer, 2009.

[12] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. DiP: A Parallel Program Development Environment, 1996.

[13] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob. The Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev.*, 38(4):37–42, Mar. 2011.

[14] G. Rodriguez, R. Beivide, C. Minkenberg, J. Labarta, and M. Valero. Exploring Pattern-aware Routing in Feneralized Fat Rree Networks. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 276–285, New York, NY, USA, 2009. ACM.

[15] V. Sarkar. ExaScale Software Study: Software Challenges in Extreme Scale Systems. Technical report, DARPA IPTO, September 2009.

[16] S. S. Shende and A. D. Malony. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[17] F. H. Streitz, J. N. Glosli, M. V. Patel, B. Chan, R. K. Yates, and B. R. de Supinski. 100+ TFlop Solidification Simulations on BlueGene/L. In *Supercomputing Conference*, 2005.

[18] V. Sundarapandian. *Probability, Statistics and Queueing Theory*. PHI Learning, 2009.

[19] Y. Vardi. Network Tomography: Estimating Source-Destination Traffic Intensities from Link Data. *Journal of the American Statistical Association*, 91(433):365–377, Mar. 1996.

[20] G. Zheng, G. Kakulapati, and L. V. Kalé. BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines. In *IPDPS*. IEEE Computer Society, 2004.